VECTOTest Framework User Guide

VECTOTest Framework User Guide

- VECTOTest Framework User Guide
  - 1. Introduction
  - 2. Create a Test Class
    - 2.1 Test Class example
  - 3. Normal test cases
    - 3.1 TestCase arguments
    - 3.2 Output
  - 4. Analyze test cases
    - 4.1 Output
    - 4.2 Example of Analyze test cases
  - 5. Lambda expressions test cases
    - 5.1 Output
  - 6. Full example
  - 7. Migration script
    - Usage

1. Introduction

VECTOTest is a *NUnit* based test framework designed specifically for the **VECTO** project.

Its purpose is to test various different cases for one VECTO simulation run. As input data, the test framework uses a `ModalResults` object populated during a simulation and a list of `TC` or `TestCase`.

The framework then goes through the `TC` one by one and uses *NUnit* asserts to verify the conditions. Should one of the `TC` fail, `VECTOTest` will output the correct `TC` in the Test output.

2. Create a Test Class

To write a new test using this framework, create a new unit test class and do the following steps:

1. Include the following namespace
   - `using TUGraz.VectoCore.Tests.TestFramework;`
2. Inherit the `VECTOTest` class

3. Provide a value for the `BasePath` class member. This path represents the folder where the `.vecto` files relevant for the tests are located. The framework will run the simulation based on the `.vecto` file found at the path

   ```
   "BasePath/<VECTOjob_name>.vecto"
   ```

4. Write `Test` functions with the signature that follow the convention:

   ```
   public void <VECTO_job_name>_<VECTO_cycle_name>()
   ```

5. Write the actual `TestCases`

2.1 Test Class example

```
using TUGraz.VectoCore.Tests.TestFramework;
class ADASCreateTestNewFramework : VECTOTest
{
    public override string BasePath { get => @"TestData/Integration/ADAS-Conventional/Group5PCCEng/"; }

    [TestCase]
    public void Class5_PCC123EcoRollEngineStop_CrestCoast2_Conventional() => RunTestCases(MethodBase.GetCurrentMethod().Name,
            TC(0, 1234, ModalResultField.dist, Operator.Greater, 0.00),
            TC(70, 80, 10, "ColumnName", Operator.Lower, new double []{8}),
            TC(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet, new double []{0, 8, 10, 12}),
            TC(70, 80, ModalResultField.v_act, Operator.MinMax, (20, 70))
            // ... more test cases
    );
    // ... more test functions
}
```

3. Normal test cases

A `TC short for TestCase` can be defined as follows:

- `TC(start, end, column, Operator, expected_value(s)), SegmentType (enum)`
- `TC(start, start_tolerance, end, end_tolerance, column, Operator, expected_value(s), SegmentType (enum)`
- `TC(start, end, time_tolerance, column, Operator, expected_value(s), SegmentType (enum)`
- `TC(lambda_expression1, lambda_expression2`

3.1 TestCase arguments

- `start` - type `double`
  - The row in the mod file that has the `dist` or `dt` column greater than or equals to this value will be the first row in which testing will be performed.
- `start_tolerance` - type `double`
  - Tolerance value that is added to `start`. Either represents distance in `meters` or time difference in `seconds`
- `end` - type `double`
  - The row in the mod file that has the `dist` or `dt` column lower than or equals to this value will be the last row in which testing will be performed.
- `end_tolerance` - type `double`
  - Tolerance value that is subtracted from `end`. Either represents distance in `meters` or time difference in `seconds`
- `column` - type `string` or `ModalResultsField`
  - The column from the `ModalResults` table that is tested.
- `Operator` - type `Enum`
  - Represents what property the `column` has to respect in order for the test to pass.
  - Possible options:
    - `Equals` to `expected_value`
    - `Lower` than `expected_value`
    - `Greater` than `expected_value`
    - `MinMax` - `column` is between `(min, max)` values
    - `ValueSet` - `column` is any of the values in `expected_values` array
- `expected_value(s)` - type `int` | `double` | `int[]` | `double[]` | `null` | tuple `(min, max)`
  - Depending on the type of `Operator`, the `expected_value(s)` argument can have different types, as follows:
    - If `Operator` is `Equals` | `Lower` | `Greater` - single value as `double` or `int`
    - If `Operator` is `MinMax` - tuple with two `int` or `double` values representing the minimum and the maximum value for the `column`
    - If `Operator` is `ValueSet` - array of `int` or `double`. The `column` value has to be equals to any of the elements of this array.
- `SegmentType` - type `Enum`
  Specifies which column `start` and `end` reference.
  Possible values:
  - `Distance` - `start` and `end` are distances

  - `Time` - `start` and `end` are time points
- `lambda_expression` - A specialized type of `TC` takes as argument two Lambda expressions. More details in a later section.

3.2 Output

For each `TestCase`, the framework will produce different outputs depending on whether or not the respective `TestCase` has failed or not.

For instance, for the `TestCase`s:

```
TC(start, end, column, operator, expected_value, segment_type)
TC(start, start_tolerance, end, end_tolerance, column, operator,
expected_value, segment_type)
TC(start, end, time_tolerance, column, operator, expected_value)
```

If passed the output is:

```
(start, end, column, operator, expected_value) -> ✔ Pass
(start, start_tolerance, end, end_tolerance, column, operator,
expected_value) -> ✔ Pass
(start, end, time_tolerance, column, operator, expected_value) -> ✔ Pass
```

However, if the `TestCase`s failed, the framework outputs the correct condition as well:

```
(start, end, column, operator, expected_value) -> ✗ Fail
    *(start, end, column , operator, correct_expected_value)
    * ...

(start, start_tolerance, end, end_tolerance, column, operator,
expected_value) -> ✗ Fail
    *(start, start_tolerance, end, end_tolerance, column , operator,
correct_expected_value)
    * ...

(start, end, time_tolerance, column, operator, expected_value) -> ✗ Fail
    *(start, end, time_tolerance, column , operator, correct_expected_value)
    * ...
```

4. `Analyze` test cases

The `Analyze` test cases are useful when the developer wants to get information about a specific field in the `ModalResults` table fast.

In general, the `Analyze` test case just tells the framework to check the value of the `column` from `start` to `end` and print out the true `TC` to the test output. The framework will throw an exception and thus the test will be marked as failed if an `Analyze` test case is present.

`Analyze` test cases are written in a similar manner to normal test cases, with the main difference that the `expected_value` **must** be `null`. `Analyze` test cases can be part of the same test function as normal test cases, and they will be handled separately by the framework.

4.1 Output

For the `TestCase`

```
TC_Analyze(start, end, column, operator, segment_type=SegmentType.Distance)
TC_Analyze(start, start_tolerance, end, end_tolerance, column, operator,
segment_type=SegmentType.Distance)
TC_Analyze(start, end, time_tolerance, column, operator,
segment_type=SegmentType.Distance)
```

a possible output would be

```
(start, end, column, operator, segment_type) analysis:
    *(start, end, column, operator, correct_expected_value, segment_type)
    * ...

(start, start_tolerance, end, end_tolerance, column, operator, segment_type)
analysis:
    *(start, start_tolerance, end, end_tolerance, column, operator,
correct_expected_value, segment_type)
    * ...

(start, end, time_tolerance, column, operator, segment_type) analysis:
    *(start, end, time_tolerance, column, operator, correct_expected_value,
segment_type)
    * ...
```

4.2 Example of `Analyze` test cases

```
TC_Analyze(0, 1e6, "DriverAction", Operator.Equals),
TC_Analyze(0, 1234, 10, ModalResultField.dist, Operator.Greater),
TC_Analyze(0, 5, 1234, 100, ModalResultField.dist, Operator.Lower),
TC_Analyze(0, 1234, ModalResultField.altitude, Operator.MinMax,
SegmentType.Time),
TC_Analyze(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet)
```

For each `Analyze` test case, the `VECTOTest` will output the correct `TC` based on the data found in the `ModalResults` table:

```
====== JOB INFORMATION ======
  JOB: TestData/Integration/ADAS-Conventional/Group5PCCEng/Class5_PCC123EcoRollEngineStop.vecto
CYCLE: CrestCoast2
 TEST: Class5_PCC123EcoRollEngineStop_CrestCoast2_Conventional

====== DETAILED RESULTS =====
(0, 1000000, "DriverAction", Operator.Equals_Analyze, 0.00) analysis:
    *(0, 0.25, "DriverAction", Operator.Equals, 0.00)
    *(0.25, 7.18, "DriverAction", Operator.Equals, 6.00)
    *(7.18, 10.44, "DriverAction", Operator.Equals, 2.00)
    *(10.44, 22.02, "DriverAction", Operator.Equals, 6.00)
    *(22.02, 27.53, "DriverAction", Operator.Equals, 2.00)
    *(27.53, 44.43, "DriverAction", Operator.Equals, 6.00)
    *(44.43, 52.07, "DriverAction", Operator.Equals, 2.00)
    *(52.07, 153.38, "DriverAction", Operator.Equals, 6.00)
    *(153.38, 166.64, "DriverAction", Operator.Equals, 2.00)
    *(166.64, 454.95, "DriverAction", Operator.Equals, 6.00)
    *(454.95, 474.32, "DriverAction", Operator.Equals, 2.00)
    *(474.32, 3614.58, "DriverAction", Operator.Equals, 6.00)
    *(3614.58, 3672.97, "DriverAction", Operator.Equals, 4.00)
    *(3672.97, 3890.85, "DriverAction", Operator.Equals, 2.00)
    *(3890.85, 3900.92, "DriverAction", Operator.Equals, -5.00)
    *(3900.92, 3910.99, "DriverAction", Operator.Equals, 4.00)
    *(3910.99, 4505.08, "DriverAction", Operator.Equals, -5.00)
    *(4505.08, 4723.19, "DriverAction", Operator.Equals, 4.00)
(0, 1234, 10, ModalResultField.dist, Operator.Greater_Analyze, 0.00) analysis:
    *(0, 1234, 10, ModalResultField.dist, Operator.MinMax, (0.00, 1038.19))
(0, 5.00, 1234, 100.00, ModalResultField.dist, Operator.Lower_Analyze, 0.00) analysis:
    *(0, 5.00, 1234, 100.00, ModalResultField.dist, Operator.MinMax, (5.52, 1125.69))
(0, 1234, ModalResultField.altitude, Operator.MinMax_Analyze, (0.00, 0.00), SegmentType.Time) analysis:
    *(0, 1234, ModalResultField.altitude, Operator.MinMax, (-3.50, 15.60), SegmentType.Time)
(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet_Analyze, new []{0}) analysis:
    *(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet, new []{0, 8, 10, 12})

✗ 0/5 test cases passed
```

```
==================
CORRECT TEST CASES

TC(0, 0.25, "DriverAction", Operator.Equals, 0.00),
TC(0.25, 7.18, "DriverAction", Operator.Equals, 6.00),
TC(7.18, 10.44, "DriverAction", Operator.Equals, 2.00),
TC(10.44, 22.02, "DriverAction", Operator.Equals, 6.00),
TC(22.02, 27.53, "DriverAction", Operator.Equals, 2.00),
TC(27.53, 44.43, "DriverAction", Operator.Equals, 6.00),
TC(44.43, 52.07, "DriverAction", Operator.Equals, 2.00),
TC(52.07, 153.38, "DriverAction", Operator.Equals, 6.00),
TC(153.38, 166.64, "DriverAction", Operator.Equals, 2.00),
TC(166.64, 454.95, "DriverAction", Operator.Equals, 6.00),
TC(454.95, 474.32, "DriverAction", Operator.Equals, 2.00),
TC(474.32, 3614.58, "DriverAction", Operator.Equals, 6.00),
TC(3614.58, 3672.97, "DriverAction", Operator.Equals, 4.00),
TC(3672.97, 3890.85, "DriverAction", Operator.Equals, 2.00),
TC(3890.85, 3900.92, "DriverAction", Operator.Equals, -5.00),
TC(3900.92, 3910.99, "DriverAction", Operator.Equals, 4.00),
TC(3910.99, 4505.08, "DriverAction", Operator.Equals, -5.00),
TC(4505.08, 4723.19, "DriverAction", Operator.Equals, 4.00),
TC(0, 1234, 10, ModalResultField.dist, Operator.MinMax, (0.00, 1038.19)),
TC(0, 5.00, 1234, 100.00, ModalResultField.dist, Operator.MinMax, (5.52,
1125.69)),
TC(0, 1234, ModalResultField.altitude, Operator.MinMax, (-3.50, 15.60),
SegmentType.Time),
TC(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet, new []
{0, 8, 10, 12}),

==================
```

5. Lambda expressions test cases

A specialized type of `TestCase` takes as arguments two Lambda expressions. This feature gives the developer great flexibility as it makes it possible to test more or less anything.

Both Lambda's take as an argument a `DataRow` object and **must** return `bool`. The test framework will check the first lambda expression over the whole `ModalResults` table, and wherever the first Lambda holds, the second Lambda must hold as well.

Example:

```
TC(
    (DataRow dr) =>
    {
        return dr.Field<System.UInt32>
(ModalResultField.Gear.GetName()).Value() >= 0;
    },
    (DataRow dr) =>
    {
        return dr.Field<SI>(ModalResultField.v_act.GetName()).Value() >= 0;
    }
)
```

5.1 Output

Because the Lambda expressions are defined by the user and there is no way of knowing what is actually tested with them, the only output provided by the framework is whether or not the test case passed or failed, and the position where the fail occured:

- Pass

  ```
  Lambda expression -> ✔ Pass
  ```

- Fail

  ```
  Lambda expression -> ✗ Fail
      * Lambda expression failed at position: <dist> m
  ```

6. Full example

In the end, the framework gathers all the correct conditions (including those that passed), and outputs them in the console to make it convenient for the developer to copy and paste the correct ones in the function and have all the test cases pass.

For the following test cases ...

```
class ADASSampleTests : VECTOTest
{
    public override string BasePath { get => @"TestData/Integration/ADAS-
Conventional/Group5PCCEng/"; }
    public void Class5_PCC123EcoRollEngineStop_CrestCoast2_Conventional() =>
RunTestCases(MethodBase.GetCurrentMethod().Name,
        TC(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet,
new []{1, 2, 3, 4}),
        TC(70, 80, ModalResultField.v_act, Operator.MinMax, (70, 150)),
        TC(70, 80, ModalResultField.altitude, Operator.Lower, 150),
        TC(70, 80, ModalResultField.v_act, Operator.Greater, 100),
```

```
        TC(70, 80, "PCCState", Operator.Equals, (int)
PCCStates.OutsideSegment),
        TC(
            (DataRow dr) =>
            {
                return dr.Field<System.UInt32>
(ModalResultField.Gear.GetName()) >= 0;
            },
            (DataRow dr) =>
            {
                return dr.Field<SI>
(ModalResultField.v_act.GetName()).Value() >= 0;
            }
        ),
        TC_Analyze(70, 80, "DriverAction", Operator.MinMax)
    );
}
```

... a possible output would be:

```
====== JOB INFORMATION ======
  JOB: TestData/Integration/ADAS-
Conventional/Group5PCCEng/Class5_PCC123EcoRollEngineStop.vecto
CYCLE: CrestCoast2
 TEST: Class5_PCC123EcoRollEngineStop_CrestCoast2_Conventional

====== DETAILED RESULTS =====
(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet, new []{1,
2, 3, 4}) -> ✗ Fail
    *(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet, new
[]{0, 8, 10, 12})
(70, 80, ModalResultField.v_act, Operator.MinMax, (70.00, 150.00)) -> ✗ Fail
    *(70, 80, ModalResultField.v_act, Operator.MinMax, (9.25, 9.58))
(70, 80, ModalResultField.altitude, Operator.Lower, 150.00) -> ✔ Pass
(70, 80, ModalResultField.v_act, Operator.Greater, 100.00) -> ✗ Fail
    *(70, 80, ModalResultField.v_act, Operator.MinMax, (9.25, 9.58))
(70, 80, "PCCState", Operator.Equals, 0.00) -> ✔ Pass
Lambda expression -> ✔ Pass
(70, 80, "DriverAction", Operator.MinMax_Analyze, (0.00, 0.00)) analysis:
    *(70, 80, "DriverAction", Operator.Equals, 6.00)

✗ 3/7 test cases passed

==================
CORRECT TEST CASES

TC(70, 2.00, 1000, 20.00, ModalResultField.Gear, Operator.ValueSet, new []
{0, 8, 10, 12}),
TC(70, 80, ModalResultField.v_act, Operator.MinMax, (9.25, 9.58)),
TC(70, 80, ModalResultField.altitude, Operator.Lower, 150.00),
TC(70, 80, ModalResultField.v_act, Operator.MinMax, (9.25, 9.58)),
TC(70, 80, "PCCState", Operator.Equals, 0.00),
TC(70, 80, "DriverAction", Operator.Equals, 6.00),

==================
```

7. Migration script

For easier conversion of old tests to the new framework, a simple Python script has been added. The script is called `migration_script_PCCTests.py` and can be found in `VectoCoreTest/TestFrameworkLib/migration_script_PCCTests.py`.

The way the script works is it reads the whole `.cs` test class, and looks for segment conditions that follow this pattern:

```
pattern = r"\t\t\t\(.*\,.*\,.*\,.*\).*"

Example of segment conditions that are matched by the script:
(0, 689, OutsideSegment, Accelerate),          // len: 689m
(689, 2066, WithinSegment, Accelerate),        // len: 1377m
(2066, 2377, UseCase1, Coast),                 // len: 311m
(2377, 2984, WithinSegment, Accelerate),
```

For each line, the script creates two lines that are compatible with the new framework and represent the equivalent condition:

```
(0, 689, OutsideSegment, Accelerate) =>

TC(0, 689, "PCCState", Operator.Equals, (int) OutsideSegment),
TC(0, 689, "DriverAction", Operator.Equals, (int) Accelerate),
```

Usage

The script accepts two input parameters:

- `path` - type `str`
  - Represents the path to the test class to be converted.
    Note: Path should be relative to the `TestFrameworkLib` folder.
- `time_tolerance` - type `float`, optional parameter

  - If provided, this tolerance will be applied to all the segment conditions. If not provided, `time_tolerance` will be 0.
    To use the script, simply pass the path of the test class , for example:

```
$ cd TestFrameworkLib
$ ./migration_script_PCCTests.py ../Integration/ADAS/ADASTestsConventional.cs 2.75
Converting file: /home/alex/tugraz/VECTO/vecto-dev/VectoCore/VectoCoreTest/TestFrameworkLib/../Integration/ADAS/ADASTestsConventional.cs
Done converting test class!
Converted file: /home/alex/tugraz/VECTO/vecto-dev/VectoCore/VectoCoreTest/TestFrameworkLib/../Integration/ADAS/ADASTestsConventional_migrated.cs
```

If the conversion succeeded, the script will produce a new file in the same folder and with the same name as the source, and append `_migrated` to its name. **The source file will remain untouched.**

Note: The script was only tested under Linux, using `Python 3.10.4` , and only shown to work for the test classes in `VectoCoreTest/Integration/ADAS`. **To use it under Windows, or for other test classes, some adaptions may be necessary.**